Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 211

Visual Comparison of Software Modularizations to Multiple Clustering Results

Jan Melcher

Course of Study:	Softwaretechnik
Examiner:	Prof. Daniel Weiskopf
Supervisor:	Dr. Fabian Beck
Commenced:	April 14, 2015
Completed:	October 14, 2015
CR-Classification:	D.2.2, H.5.2

Abstract

Modularization is a central aspect of software quality, a well-structured hierarchy of classes and packages crucial for maintainability. There is not one perfect package structure, but several measures exists that suggest how a good modularization could look like. The key is to compare these suggested hierarchies to the current hierarchy to improve it.

This thesis presents a visualization for comparing one hierarchy to a set of alternative hierarchies of the same underlying items. It is based on icicle plots that are drawn as small multiples. Inner nodes are color coded according to their similarity to nodes of the other hierarchies. The user can select arbitrary sets of leaf nodes to further inspect their cumulative similarity to the nodes of all hierarchies. The visualization is put in context of software modularization by using its terminology and providing features such as to show the source code. It aims to be easy to use once the idea is explained and invites the user to explore the data set.

Kurzfassung

Modularisierung ist ein zentraler Aspekt der Softwarequalitätssicherung. Eine gut strukturierte Klassen- und Pakethierarchie trägt in besonderem Maße zur Wartbarkeit eines Systems bei. Aus verschiedenen Metriken können verschiedene Paketstrukturen abgeleitet werden und keine davon hat den Anspruch, die einzig korrekte zu sein. Aus dem Vergleich der generierten Hierarchien mit der existierenden Paketstruktur können Verbesserungsvorschläge abgeleitet werden.

In dieser Ausarbeitung wird eine Visualisierung vorgestellt, mit der eine Hierarchie mit einer Menge anderer Hierarchien derselben Klasse verglichen werden kann. Sie basiert auf dem Icicle-Plot-Ansatz und zeigt die Bäume als Small-Multiples-Visualisierung nebeneinander an. Innere Knoten sind entsprechend ihrer Ähnlichkeit zu Knoten der anderen Hierarchien eingefärbt. Der Benutzer kann beliebige Mengen von Klassen auswählen, um deren kumulative Ähnlichkeit zu anderen Knoten genauer zu untersuchen. Die Visualisierung ist durch die Terminologie und spezielle Funktionalität wie die Quellcodeanzeige in den Kontext der Softwaremodularisierung eingebettet. Sie soll für Benutzer, die mit den Konzepten vertraut sind, einfach zu bedienen sein und dazu einladen, den Datensatz zu erforschen.

Contents

List of Figures

List of Listings

1 Introduction

Large software systems consist of thousands of modules, each representing a small piece of the whole application. In modern object-oriented programming languages, these modules are in turn organized in a hierarchy of packages (or namespaces, like they are called in other languages). Determining which modules should belong to which packages is a responsible task because it affects how well the software is understood and thus has an influence on maintainability.

There are several principles that should be respected when structuring of software modules, such as high cohesion and low coupling, information hiding, or Conway's Law. It is possible to deduce concepts for coupling between modules and thus create a graph. By applying a clustering algorithm on this graph, a package structure that complies to one of the concepts can be automatically generated.

However, the different coupling concepts often contradict each other. They reflect guidelines rather than instructions for creating a good package structure. And, most importantly, modularizations grow slowly and the developers would be highly confused if the structure changed overnight. Instead, to improve the package structure of an existing application, the automatically generated clusterings should be compared to the existing modularization. A developer then can identify similarities for parts of the modularization that follow one or more principles or differences where the structure may be improved.

To support this comparison, a visualization is to be developed. It should be based on the existing package structure to give the user comfort of the familiar. For each package, it has to become evident how similar it is to the generated clusterings. Packages with good matches in the clustering hierarchies should be easily spotted. Ideally, a closer inspection reveals the reasoning behind packages and thus provide insights that would not be apparent from just looking at the source code.

The visualization presented in this work assists software developers in several ways. The first one is refactoring code by restructuring the packages and has already been touched on. The visualization provides support for packages that are already backed by a known metric. It also hints at packages with low resemblance to any clusters. When those are inspected, the visualization shows whether modifications of the package, such as through union or division, would improve the similarity to clusterings.

Secondly, the visualization aids a developer who is new to a software project in deciding where to put a new class. If one is not yet comfortable with the package structure, it may be hard to find out how exactly each package is defined. The visualization tool gives hints at which rules have been followed creating certain packages. This provides more confidence in choosing the right package than just looking at the name and glancing over the contained classes.

More generally, the visualization assists in understanding source code by pointing out which classes belong to each other in which sense. The special focus on package structure supports analysts by providing a known structure as guide in a field of new information.

1.1 Coupling Concepts

The data used in the visualization is based on six different concepts of code coupling which will be briefly introduced here. This collection has been adopted from Beck and Diehl [BD11].

1.1.1 Structural Dependencies

Stevens et al. [SMC74] have introduced the concept of code coupling, the degree to which modules depend on each other, and cohesion, indicating how much the parts of are related. Low coupling is a sign of solid modules which can mostly work on their own and thus can be reused and changed independently. A system of highly coupled modules, on the other hand, is hard to maintain because altering one module likely requires adjusting other modules, too. The cohesion metric indicates whether a module could be split up while still preserving low coupling. If there are multiple independent clusters in one module, i.e., the cohesion is low, the module likely serves two purposes and should be divided.

Structural dependencies are a metric for measuring coupling and cohesion. The dependencies between classes in the same module should be high whereas those between classes of different modules should be low.

We will use three definitions of *dependency* [BD11]:

- **Usage**: A class depends on another one if it refers to it or one of its methods in a method body.
- **Aggregation**: A class depends on another one if it has a field of its type.
- **Inheritance**: A class depends on another one if it extends it.

1.1.2 Fan-Out Similarity

A second principle for good software modularization is Information Hiding, introduced by Parnas [Par71]. He claims that a good programmer uses all information available about the system and dependencies to write efficient programs. However, this information is based on design decisions that may change over time. Therefore, he advocates hiding this information from the user, so that the dependent modules still work when design decisions of dependencies change. Parnas also suggests to use this principle as criterion for software modularization [Par72].

Schwanke [Sch91] extends this idea and provides an “information sharing heuristic”: Classes that use the same modules are likely to share design decisions. Beck and Diehl [BD11] apply this heuristic to the three definitions of module dependency as given above. A feature vector that represents the dependency of one class to each other class is created. These feature vectors are compared using the cosine similarity measure so that the vector lengths, i.e., the absolute numbers of dependencies of a class, are not taken into account. Classes with similar feature vectors are assumed to share common information and thus are coupled.

For each of the three dependency definitions, two *fan-out* coupling measures are generated: one using dependencies to internal classes, the other one taking only external libraries into account.

1.1.3 Evolutionary Coding

Information hiding allows local changes of modules without having to alter classes of other modules. This implication can also be reversed: Classes that often have to be touched at the same time should be grouped in a module. The concept of evolutionary coding uses the history of source code changes to identify classes that are often changed together. We use two variants: The support value which represents the absolute number of common changes and the confidence value which is normalized to the total number of changes of the first class. These information are taken from the project's version control repository. Details are described by Beck and Diehl [BD11].

1.1.4 Code Ownership

Conway's Law [Con68] states that the organization structure is at some point reflected by the structure of a software system. This means that classes that are maintained by the same group of developers tend to be grouped in the same modules. Following this train of thought, designing a package structure that does not respect this principle would be pointless because future, natural, changes to the modularizations would introduce it again. Additionally, it allows developers to familiarize themselves with modules they have to work with and hides parts of the system they do not need to interact with.

Like evolutionary coding, the code ownership coupling can be derived from source control. Once again, feature vectors and the cosine similarity are used. We use two variants, one where the number of changes to a class is taken into account, and one where just the fact that a person changed a class matters.

1.1.5 Code Clones

While code clones should generally be avoided, sometimes they serve a valid purpose such as performance improvement or dependency reduction. In either case, if the code clone had been avoided, there would be a different kind of coupling. For example, if a method call is inlined into two classes, the two method calls are removed and thus the fan-out similarity is avoided [BD11]. For this reason, code clones are used as coupling concept. Code Clones are are classified in four types [RC07], of which the first two have been chosen. Type I clones are identical pieces of code, ignoring indentation and code layout. Type II allows renaming of identifiers.

1.1.6 Semantic Similarities

This coupling disregards the special syntax of source code and instead only takes the actual English words into account. Standard text similarity measurements such as tf-idf (term frequency, inverse document frequency [Cho10]) can be used to measure the similarity of two classes. Preprocessing includes removing license texts and splitting camel case identifiers into their parts.

1.2 Clustering

All these coupling concepts can be formalized into functions that take two classes as arguments and map them to a coupling value. This value is normalized to the range from 0 to 1.

As both arguments are discrete and finite, a similarity function can also be represented by a graph. The graph contains one node for each class. Every pair of classes that, when being applied to the coupling function, results in a non-zero coupling value is connected by an edge; the coupling value being the edge weight.

By analyzing these graphs, clusters of highly-connected nodes can be found. A clustering algorithm finds recursive clusters of nodes and thus generates a hierarchy from the coupling values. Now, the underlying graph structure can be discarded for the most part. The hierarchies contain the useful information of class clusters in condensed form. Only for detailed inspection, the original graphs or coupling values might become useful again.

1.3 Asymmetric Comparison

After the clustering process, there are several hierarchies – one for the original package structure, and one for each clustering. They all have the same set of nodes, i.e., the classes, but the edge set is different. The purpose of the visualization is to show the differences between the edge sets of the visualizations.

There are two kinds of comparisons: symmetric and asymmetric. In a symmetric comparison, all objects to be compared are considered equal, they are all compared to each other. This is the most general approach and would show the most information. However, in our case, the hierarchies are not equal: The original structure takes a special place as the reference hierarchy. Therefore, an asymmetric comparison is used. The reference hierarchy, also called *primary hierarchy*, is compared to all the hierarchies derived from the clustering algorithms, called *secondary hierarchies*.

Outline

The rest of this thesis will be structured as follows:

Chapter 2 – Related Work presents existing approaches to visually comparing hierarchies.

Chapter 3 – Visualization introduces the visualization, explains its details and justifies design decisions.

Chapter 4 – Implementation gives an overview over the architecture of the visualization tool, technologies and technical design decisions.

Chapter 5 – Evaluation features a small case study that uses the visualization to explore the package structure of two open source software projects.

Chapter 6 – Conclusion summarizes the work of this thesis, compares the evaluation results to the goals introduced in the introduction and proposes future work.

2 Related Work

Tree visualization is a common task across a wide range of domains due to the abstract and reusable concept of trees. Tree comparisons, on the other hand, are much less researched. Comparisons of multiple trees are especially rare to find.

There is however one science field with a task very similar to ours. Systematic biology tries to find the correct evolutionary trees of all the species. Observed information, e.g., DNA sequence data, are used as input for algorithmns that try to find evolutionary trees that explain these observations. However, there is not one such resulting tree but many, and they must be considered in total to derive substantial theories about the evolution [AK02].

While this description is surely oversimplified, it shows how closely related systematics in biology is to software modularizations. Species are the equivalent to classes. Observations about creatures and how they can be found in multiple species correspond to the coupling concepts we have defined for related classes. Species that share a lot of DNA sequences are considered *close* to each other the same way as classes are related when they, e.g., share code clones. The algorithms trying to explain observations about species in terms of evolution are, in their kind, similar to the clustering algorithm that turns the coupling matrices of classes into hierarchies. Thus, many tree comparison visualizations are created for application in biology.

Graham and Kennedy [GK10] have surveyed and classified existing visualizations for trees as well as comparisons between two and more trees. This chapter will describe those that most closely fit into our task.

2.1 TreeJuxtaposer

TreeJuxtaposer [MGT+03] is a tool from the context of systematic biology. It can be used to compare multiple trees, but only two at a time. The two trees are displayed side-by-side as node-link diagrams with rectangular layout. To support the user in the comparison process, node links can be colored in different ways, most of them depending on a node similarity value. This value is defined by the number of leaf nodes shared between two inner nodes: Nodes with roughly the same set of leaf nodes are considered similar. Next, the *best corresponding node* of a node A is defined as the node of the other hierarchy with the highest similarity to node A.

There are four modes that determine how nodes are colored. In the first color mode, when the user select a node in one hierarchy, the best corresponding of the other three is highlighted. Also, the leaves of the selected node are enclosed in a rectangle to highlight the extend of the selected node. The second mode is a simple search for a node by known by name. The third color mode allows to find structural differences by coloring all nodes that do not have an exact corresponding node in the other hierarchy red. In the fourth mode, the user can select a subtree, and all nodes in this subtree as well as the best corresponding nodes of the selected nodes are highlighted in a user-specified color.

These features allow an efficient comparison of two trees. Automatic detection of best corresponding nodes provides great support and leads the user to points of interest. The interaction features can be used to inspect individual subtrees and thereby support or refute theories about structural differences.

2.2 DoubleTree

DoubleTree [PLCB04] also has been created for biologists. The application divides its main window vertically into two areas that each display one of the hierarchies. For tree visualization, they use a node-link diagram.

Comparison is a very interactive process in this tool. By default, both trees are collapsed. Nodes can be expanded by clicking on them. This also highlights the corresponding node in the second hierarchy. If the selected node maps to several nodes, they are all highlighted.

The visualization is suitable for exploring deep hierarchies. By looking at and following highlighted nodes in the second hierarchy, it becomes clear which nodes have very similar matches in the second tree, and which are mapped to multiple nodes or to none. The user can search for nodes by name, which then are expanded in the first hierarchy. That way, the path to said node and the path to the corresponding nodes can be compared.

The authors decided against using lines to link corresponding nodes of both hierarchies because it would generate too much clutter. These lines could be confused with those representing parent-child relations as part of the node-link diagram. Instead, only a generic background color for matched nodes is used.

The visualization could theoretically be scaled up to multiple trees, by dividing the window into multiple areas each containing one tree. The interaction could be mapped directly – selecting a node in the first hierarchy could focus corresponding nodes in all the other hierarchies.

However, it only compares nodes on a local basis. The tool never considers whole trees. In fact, when the user switches to a node on a different branch, the other branch is closed – only nodes in one straight path from the root to an inner or leaf node, and their direct children, can be viewed at once. This seriously limits its use at comparing moderately sized trees with no specific goal in mind.

2.3 Linked Icicle Plots

Holten and Van Wijk [HVW08] present a visualization for two trees. They are displayed as icicle plots that grow from the top down and from the bottom up, respectively. At places where the depth is smaller, nodes are stretched vertically. This evens out the inner surface and a rectangular space is left in between the two trees.

In this space, links between leaf nodes are drawn. In the most basic mode, the links go in a straight line between corresponding nodes. This only shows which leaf nodes have moved; structural change is hard to identify. The power lies in edge bundling, which can be toggled for individual layers of both hierarchies. The stronger edges are bundled, the better structural changes are depicted. Following the flow of an edge bundle, one can easily see if it splits in multiple nodes at the other hierarchy or if multiple nodes are merged.

The user can interact with the visualization by drawing a line through the bundled links. This selects all edges crossing the line. Nodes completely covered by the selection are highlighted in green; those partially selected in blue. The visualization then automatically zooms into the selection.

This approach would fit very well to a comparison of two class hierarchies. In contrast to TreeJuxtaposer and DoubleTree, this visualization always displays all nodes in a compact manner. Finding corresponding leaf nodes and inner nodes would also be possible using the different edge bundling levels.

Nonetheless, it is not suitable for a comparison of multiple trees. In principle, one could add icicle plots at the left and right border, or even arrange more icicle plots in a polygon shape. But there would still only be one central area for the inter-tree edges. Even if, as in an asymmetric comparison, each tree would only need links to one primary hierarchy, there would still most certainly be too many edges to display.

2.4 Multiple Trees

There are several visualizations showing trees as small multiples. Telea's and Auber's *Code Flow* [TA08], for example, visualizes source code changes with a set of horizontally distributed tree visualizations. Between each two adjacent trees, edges connect correlating nodes, forming a parallel coordinates diagram. Via coloring, changes can be traced across the whole set of trees.

This approach however is only useful if there is some kind of ordering between the trees. In the previous example, as in many others, this ordering is time. In our case, there is no ordering, so drawing edges between adjacent trees would be arbitrary and it would not be possible to compare one tree to all others.

Graham and Kennedy [GK05] use multiple icicle plots stacked vertically. To detect structural change, they look for nodes with changed parent nodes. Interaction allows to focus on subtrees. This visualization is a good fit to compare very similar hierarchies, as it focuses on individual movements. In our case, however, there are often very large structural differences.

2.5 Summary

While there are existing visualizations for comparison of multiple trees, they do not fit well for the given task of comparing package structure to clusterings. Holten's linked icicle plots would be great for a comparison of two package structures because it highlights structural changes on a large scale well, but it does not scale to multiple trees. TreeJuxtaposer could be applied to multiple trees but does not offer interaction to support the user in aggregating information from all trees at the same time. Parallel coordinate visualizations only work well for ordered sets of trees.

However, several lessons have been learned. Edge drawing between multiple trees generates too much clutter; color mapping is a better fit. Small multiple diagrams of a condensed tree visualization like icicle plots allow to display trees of a few hundred nodes at once and thus can give a good overview. And it is important that the tool calculates a similarity metric so that it can actively aid the user to structural changes and interesting parts.

3 Visualization

In this chapter, the visualization will be presented. After introducing the approach and describing the essential parts of the visualization, design decisions will be discussed in detail and possible alternatives presented.

The application consists of three layout sections (see Figure 3.1): a header that allows to choose a software project, the content pane containing the icicle plots, and a sidebar for further information.

The whole application is depicted in Figure 3.2. In the content pane, all selected hierarchies are displayed side-by-side as icicle plots, forming a small-multiples visualization. The primary hierarchy, called *packages* because it resembles the original package structure, is always at the very left. This emphasizes its special role in the comparison. The other icicle plots visualize one clustering hierarchy each. The user can choose which clusterings are shown and rearrange the chosen ones by drag and drop (see Section 3.8 for details). Each icicle plot has a heading labeling it with the cluster identifier (or *packages* for the primary hierarchy). The background color of the heading as well as the icicle's border color are generated rom the cluster identifier and provide a subtle visual grouping of clusterings of the same category (see Section 3.10).

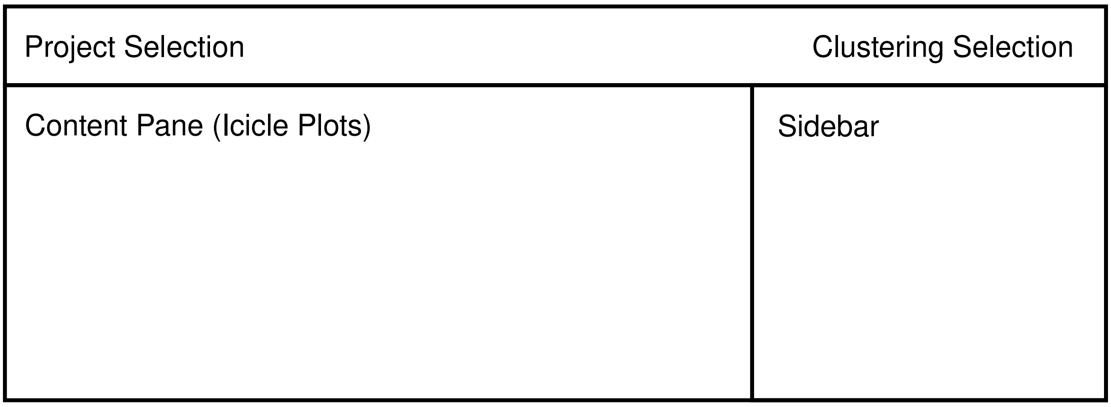
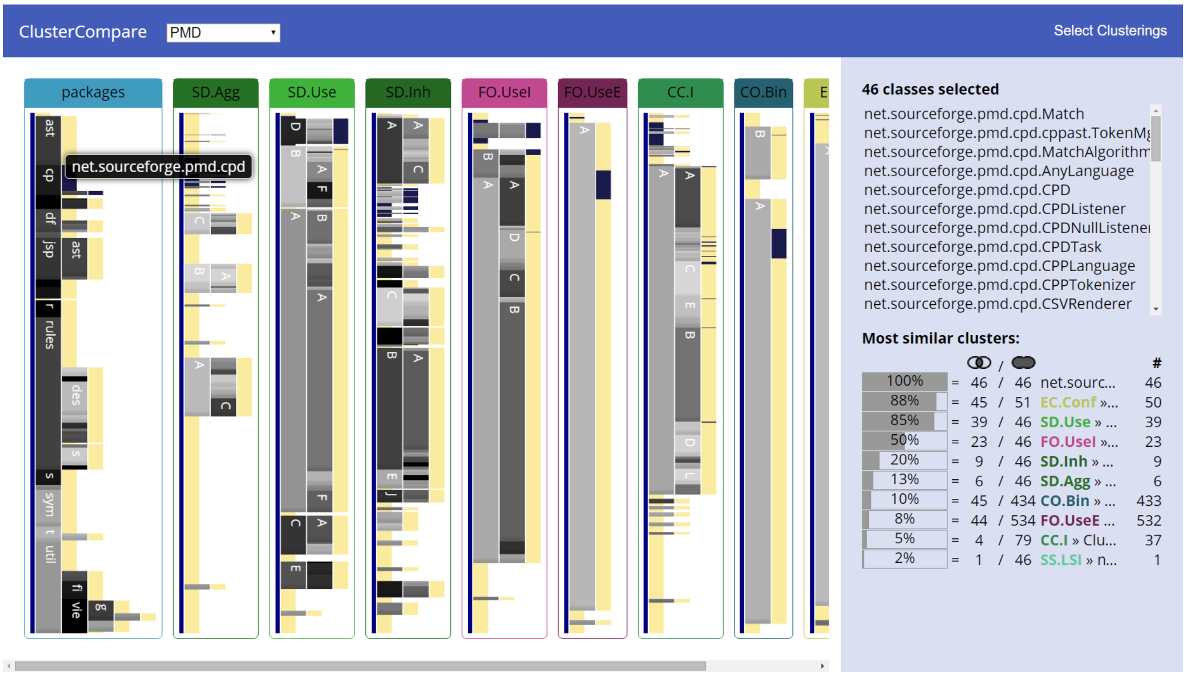


Figure 3.1: Application Layout



46 classes selected

net.sourceforge.pmd.cpd.Match
net.sourceforge.pmd.cpd.cppast.TokenMg
net.sourceforge.pmd.cpd.MatchAlgorithm
net.sourceforge.pmd.cpd.AnyLanguage
net.sourceforge.pmd.cpd.CPD
net.sourceforge.pmd.cpd.CPDListener
net.sourceforge.pmd.cpd.CPDNullListener
net.sourceforge.pmd.cpd.CPDTask
net.sourceforge.pmd.cpd.CPPLanguage
net.sourceforge.pmd.cpd.CPPTokenizer
net.sourceforge.pmd.cpd.CSVRenderer

Most similar clusters:

			/			#
100%	=	46	/	46	net.sourc...	46
88%	=	45	/	51	**EC.Conf** »...	50
85%	=	39	/	46	**SD.Use** » ...	39
50%	=	23	/	46	**FO.UseI** »...	23
20%	=	9	/	46	**SD.Inh** » ...	9
13%	=	6	/	46	**SD.Agg** » ...	6
10%	=	45	/	434	**CO.Bin** » ...	433
8%	=	44	/	534	**FO.UseE** ...	532
5%	=	4	/	79	**CC.I** » Clu...	37
2%	=	1	/	46	**SS.LSI** » n...	1

Figure 3.2: Overview of the Application

In the icicle plots, there are three different kinds of nodes: the root node, painted blue; inner nodes, colored according to similarity values (see Section 3.3); and the leaf nodes representing classes, whose color depends on the selection status (see Section 3.6). The ordering of the primary hierarchy is alphabetical; the nodes in the secondary hierarchies are rearranged to reflect the ordering of the primary hierarchy as close as possible (see Section 3.7)

The sidebar initially explains the functionality of the application in a few words. As soon as a selection has been made, it lists the selected class names and provides access to the source code (see Section 3.11). Below, a table lists the clusters that most closely resemble the selection. The table will be explained in Section 3.9.

3.1 Single Tree Visualization

Before thinking about comparison, the representation of single trees has to be chosen. Graham and Kennedy describe four basic principles of how trees can be presented: linking, indentation, stacking and nesting [GK10]. All of them are well-studied and therefore only briefly introduced here. In the following, it is outlined how well they fit the given task.

3.1.1 Node-Link Diagrams

In node-link diagrams, the root node, all inner nodes, and the leaves are represented by shapes that are connected by lines. They are intuitively understood – in this context, classes would be attached to their packages, and those to the enclosing packages. Also, it is easy to find out which nodes are connected, i.e., which classes belong to which package, just by following the links.

There are several ways to lay out a node-link diagram. Comparisons of long and wide hierarchies benefit from a predictable and structured layout. Rectangular or slanted layouts fulfill this property by ordering the nodes hierarchically in one direction. On the one hand, leaf labels can be aligned at the far end of the visualization, so their labels can be scanned easily. This would be a real advantage for the given task. Inner node labels, on the other hand, are hard to place without causing occlusion with the links – and after all, the inner node labels in our visualizations are the package names, one of the most important texts because the user is familiar with them. Additionally, the diagram would take at least as much height as the number of classes multiplied with font size. Class names could be omitted, but then the diagram would be simple a grid of connected lines and hard to perceive.

There are also circular layouts which, depending on the depth of the hierarchy, may scale better because labels are arranged on two axes. This is equally a disadvantage in the context of comparing nodes in a small-multiples visualization because both axes are used up for the one visualization and there is no natural axis on which multiple instances could be aligned and still be easily compared.

3.1.2 Indentation

Links between the nodes generate a lot of visual clutter and take away space that could be used for the nodes themselves. Indentation-based tree visualizations encode all information about the hierarchy in the node placement instead. They are known to computer users because file browsers usually display directory structures as indented tree views. IDEs also use them to visualize package structures, which makes them a good candidate for this visualization.

They are very clean: there is no clutter and no redundancy. Indentation trees also scale well: if the space suffices to display all the nodes' names, the nodes can be condensed to small rectangles, even to the size of one pixel, as Burch et al demonstrate [BRW10]. The colors of these rectangles or pixels can still be used to encode information on a per-node basis.

Finally, the layout algorithms are really simple and fast, a whole tree can be rendered in linear time.

3.1.3 Stacking

Stacking tree visualizations are very similar to indentation-based graphs in that they do not have connecting lines and also indent nodes depending on their depth. The difference lies in the rendering of inner nodes: they do not have a place on the axis among the leaves but instead take up the space that has been created for the indentation. The advantage is that less space is wasted and inner nodes get more space. While leaf nodes can be scaled down to few pixels, inner nodes are large enough to display their labels, i.e. the package names.

This visualization is not as intuitive as the node-link diagram, and is not as commonly in use as the indentation-based visualization. This means that users have to take a few moments to familiarize themselves with it. However, due to its simplicity, it can be learned fast and interpreted efficiently afterwards.

Rectangular stacking plots are called icicle plots [KL83]. There are also radial ones (called *sunburst*), but they have the same problem as radial node-link diagrams concerning small-multiple visualizations.

For this thesis, icicle plots have been selected because they scale best and offer a unique shape for each tree that can be recognized and compared.

3.1.4 Nesting

Taking the space compression one level further, nodes can also be drawn *inside* their parents. Treemaps [JS91] are an example of this approach. The tree can be fitted into any rectangular shape perfectly with no extra space. However, it is not trivial to arrange the nodes in a way that the rectangles stay mostly square. It is possible, but the arrangement now depends on the aspect ratio of the available space and thus the appearance changes drastically even when the data does not. This is not ideal for comparisons.

3.2 Comparing Hierarchies

After having chosen how to present a single hierarchy, a concept for comparing multiple hierarchies is needed. This section will summarize the tree comparison approaches as classified by Graham and Kennedy [GK10].

3.2.1 Animation

Given the abilities of multimedia, using time as dimension for displaying change seems reasonable. Indeed, the change between two hierarchies could be made clear by visually moving nodes within the tree structure. In our case, no nodes are added or removed, so there would be no appearing or disappearing nodes, and watching the movement of objects would be intuitive.

However, to fully understand the change, the animation would have to be repeated over and over. It is easy to lose track of hundreds of moving objects. Nodes would occlude each other. While it might still technically work for comparing two hierarchies, this approach completely falls short for a comparison with multiple hierarchies. There is no natural ordering of the modularizations, so animating from one visualization to the next would not make sense.

As we have chosen a very compact tree visualization before, there is room to display multiple trees at the same time, so there is no need to use the time as further dimension.

3.2.2 Agglomeration

Agglomerating trees results in a graph that contains the nodes and edges of all the trees, the edges being colored according to the tree they come from. This minimizes the redundancy as the nodes that occur in multiple trees only exist once in the agglomeration graph. However, the result is a graph and not a tree, so the only applicable one of the presented tree visualization methods is a node-link diagram, which has been ruled out for not being scalable and wasting too much space. An agglomeration of several trees with a few hundreds of classes each would generate a huge graph with little structure. However, there would be one advantage: When a cluster is exactly the same as a different cluster or a package, it is drawn as one single node with edges in multiple colors. Clusters that are very similar but not exactly the same would still be rendered as two nodes with no special connection.

3.2.3 Small Multiples

This approach has been invented by Tufte [Tuf83]. The trees are simply put side by side and presented as if they were separate trees. They can be inspected independently and compared to each other. Here it is crucial that the shapes of the individual visualizations are easily perceived. Moreover, their structure should ideally be aligned to the horizontal and/or vertical axis. To provide two counterexamples, force-directed node-link diagrams and radial stacked charts are not aligned. Tracking features between multiple instances of these charts is hard because there are no visual guidelines to follow. Rectangular stacked charts, on the other hand, are perfectly aligned with both axes. Jumping between several of them is easier because the horizontal lines provide a guide whereas the vertical lines cleanly separate the visualizations.

There are ways to support the user comparing the trees. One would be to draw links between identical nodes in the different trees, similar to parallel coordinates [ID90]. Again, as for node-link diagrams, the intuition would be met, the idea of equal things being connected is easily understood. However, like for parallel coordinates, there would be visual clutter and occlusion. Edge bundling algorithms can be used to compensate this flaw but the exact linking information would be lost. The inherent problem with this approach is that it does not work well for more than two trees because there is no sensible way of drawing the edges between two non-adjacent trees.

As a general solution, and without adding anything that would introduce clutter, we can use color to assist the comparison. Nodes can be colored depending on the similarity to other nodes. The specific color function chosen for our visualization will be discussed in the next section.

However, while a static color for each node eases the process of finding nodes that are similar to other nodes, it does not enable the user to identify those other similar nodes. More specifically, it is hard to even find one node of one tree in a different tree. To compensate for this, a brushing-and-linking technique can be used. When one or multiple nodes of one tree are selected, these nodes are highlighted in the other trees. This obviously solves the problem of finding a single node in the other trees, and it provides a foundation for finding similar inner nodes because it shows where there are clusters of highlighted leaf nodes in other trees.

As the animation, agglomeration, and edge-drawing approaches have been ruled out, the small multiple visualization with colored nodes is chosen for comparison.

3.3 Node Similarity

To be able to color-code nodes according to their similarity, a color mapping needs to be specified. For an asymmetric comparison like this one, the formula depends on whether the node is in the primary or in a secondary hierarchy. The primary hierarchy in this case is the existing package structure and the secondary hierarchies are the clustering results.

Two clusters are considered similar if they contain mostly the same classes. We could take subclusters into account, so that two clusters would only be similar if they contained the same classes and were *structured* similarly. To evaluate the effective difference between these two definitions, we have to consider four cases:

- Comparing an innermost package to an innermost cluster. Both definitions lead to the same result as there are no nested clusters to consider.
- Comparing an innermost package to a cluster containing clusters. The structural difference between the nodes would decrease their similarity even if both of them contained exactly the same classes. Thus, even if a cluster is found that exactly matches one of the packages, but further splits it up into several subclusters, that cluster would be harder to find because of the decreased similarity value. However, the cluster would provide substantial information because it does not only confirm that said package is good, but further provides a suggestion to split it up. This information should be as visible as possible.
- Comparing an innermost cluster to a package containing subpackages. This case is similar to the last one. If a cluster has been found that contains exactly the same classes like a package, but that package is in turn split up into several subpackages, this should not decrease the similarity value. The cluster still encourages keeping the package, although it cannot confirm that the subpackages are useful. The subpackage split might be motivated by a different coupling concept.
- Comparing a package with subpackages to a cluster with subpackages. If both of them are structured the same way, their similarity is rightfully high. This would be an especially good find. It would be appropiate to assign a higher similarity value to clusters with the exact same subcluster structure than to those which just share the same classes.

Only in the last case, including structure in the similarity definition would be useful. Looking at the data, clusters matching single packages is already relatively rare; one that matches it in structure is extremely uncommon if it exists at all. For this reason, only the leaf nodes should affect node similarity.

We use the approach suggested by Beck and Diehl [BWB+14]. The similarity between two nodes is the overlap of their leaf nodes, as defined by the Jaccard coefficient in Equation 3.1.

$$\text{(3.1)}\quad \mathrm{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

For clusters of the secondary hierarchies, the most similar package is determined; the cluster's node value is then the similarity to that package [BWB+14]. For the primary hierarchy, there is a special case because it has to be compared to multiple hierarchies. Instead of choosing the most similar node form one hierarchy, the most similar cluster of *all* secondary hierarchies is used as reference, and the node value is again the similarity to that cluster. This allows to spot packages that have a corresponding cluster (as well as packages without one). However, it provides no direct indication *which* cluster is the similar one. To compensate this, brushing and linking (Section 3.6) can be used, and clicking on a node reveals similar clusters in the sidebar (Section 3.9).

3.4 Node Coloring

The similarity value is mapped to a grayscale from light gray (no similarity) to black (complete overlap). White is used as background color and thus the nodes should not be completely white. A scale is not required because the visualization is only intended to give an intuitive impression. Color is not ideal to communicate exact values [Mac86]. These will be presented differently, as shown in Section 3.9. Figure 3.3 shows a close-up of nodes with different colors.

This color scale only applies to inner nodes. Root nodes always have the similarity value 1 to other root nodes because all hierarchies consist of the same classes; the same applies to leaf nodes for the same reason. It would not provide any information to paint them black, and furthermore, that would distract from black inner nodes which are of interest.

The root nodes do not convey any information whatsoever, but if they were left out, an integral part of the tree would be missing and thus the icicle plots harder to recognize. Therefore, root nodes are still drawn but only as blue thick lines.

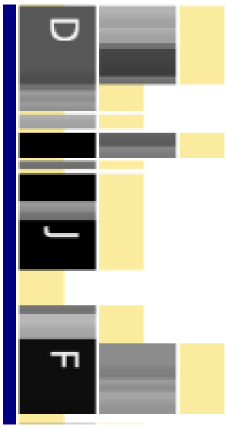


Figure 3.3: Node Coloring: The nodes *F* and *J* as well as the two black nodes above have high similarity to package nodes in the primary hierarchy. Nodes in beige are class nodes.

Leaf nodes convey important information because after all, they represent the classes that are to be structured. They do not have a similarity value, so their color can be used for further information. Section 3.6 will describe how selection information is mapped to their colors.

It would be useful to show this information at inner node level, too, e.g. highlighting packages that have all classes selected. But considering the fact that nodes can get small (sometimes even inner nodes are just one pixel high), adding an additional, separately colored region would introduce too much information and disturb the clean presentation of icicle plots. As only brightness is used to encode node similarity values, hue or saturation could theoretically encode this information. But while hue and brightness can independently be processed preattentively, they still interfere each other [Cal84].

3.5 Visualizing Borders

As the whole plot consists mostly of rectangles drawn next to each other, it is especially important to indicate where one rectangle ends and the next one starts. Otherwise, it would not be possible to distinguish nodes from each other. The easiest and most visible way of highlighting borders is by leaving space between the rectangles. This approach is used for vertical borders. All nodes have the same width (with the exception of leaf nodes, but this does not change the layout), so the horizontal gaps between the columns are nicely recognizable as white lines.

However, this method cannot be applied to vertical spacing. There are several hundreds of classes in most projects, so each leaf node is only about one pixel in height. There is no space for a one pixel gap between each leaf, and even if there would be two pixels in height available for each leaf node, every other pixel being white would result in a sparsely filled tree, so that the impression of alternating depth could arise. Even the inner nodes can be very thin, some only containing one leaf node.

Depending on the rendering technique, using vertical gaps anyway could have different results. If coordinates of rectangles are rounded before painting, some gaps and some nodes would be missing, as either the rectangle would round to a height of zero, or the next rectangle would be rendered directly after the previous because of rounding errors. So, some nodes are visible and others are invisible, where the exact pattern has no significance in relation to the actual data. This problem is called alias effect.

The alternative is to use opacity for sub-pixel rendering: If an element is to be painted in half of one pixel, the color of the element is only applied with opacity of 50% to that pixel. This is similar to how an image would be scaled down with interpolation. This solves the alias effect problem, but introduces a new one: nodes appear blurry and borders are undefined. This is not acceptable, either.

Therefore, spacing or borders cannot be used to separate rectangles vertically, at least not universally. In general, the background color could be alternated between adjacent nodes to clearly distinguish them. In our visualization, the background color is already mapped to node similarity and selection as described in the previous section. However, this color mapping already serves the purpose of node separation well in most cases: Unless two adjacent nodes have exactly the same similarity value, they are colored differently and thus can be distinguished.

Another option is to use an inset shadow in the rectangles in the bottom, by drawing a small and subtle gradient from the normal background color to a slightly darker color. This does not introduce the problems of a strictly one pixel high spacing between rectangles because it never exceeds the height of a node. There is also no alias effect because the shadow only slightly differs from the main background color – for one or two pixel high rectangles, it is hardly noticeable. In rectangles with more than a few pixels in height, however, the inset shadow clearly sets the nodes apart. This method has been chosen for inner nodes because it can be applied consistently to rectangles of any height.

Leaf nodes are mostly about one pixel in height, so using the inset shadow would be of little use. However, differentiating between leaves is not necessary: All the leaf nodes have the same height, are unlabeled and if not selected do not have any directly visible information attached to them. To reduce the clutter, leaf nodes are thus uniformly filled with a solid color.

3.6 Brushing and Linking

As we have seen, coloring gives a good overview but does not provide all required details. The most important question that cannot be answered just by looking at a set of colored icicle plots is which two nodes are actually similar to each other. Always showing this information for all nodes would result in a lot of visual clutter. Therefore, user interaction is required.

The general idea behind brushing and linking is that the user can select items they are interested in, and the visualization highlights other occurrences of these selected items [BC87]. In our case, items of interest are classes, packages, clusters as well as the whole package tree and clustering trees. To cope with these different kinds of objects, a selection strategy has to be developed.

The first idea was to expose exactly these object types to the user. They could click on a package to select the package object, or brush over some classes to select them. The tool then could show information about the selected objects, their contents (if applicable) and their relation to other objects, for example, similar clusters.

Selecting a package would be similar to selecting all the classes it contains, and if these actions would produce a different output, that might be difficult to explain. There would also be an even more complicated case: should it be possible to select objects of different types at once? Disallowing this would be inconsistent, as multiple objects of the same type could be selected (at least for classes, this is crucial, e.g., to select part of a package). But if it was allowed, what kind of detailed information should be presented?

The most tricky part about this approach, however, would be to actually work with the selection. An example workflow can demonstrate this: The user selects an interesting cluster as starting point. It looks good, however, there are some classes that do not fit in there, and therefore, the similarity results are not as accurate as they could be. The user now wants to manually remove these classes to see how the similarity information changes. The selection is now "Cluster A without classes 1, 2, and 3". Now, the user spots a different cluster that contains some additional classes that would fit well to the cluster. The further the user goes into building a good package, the more complex the selection description gets. At some point, keeping track of all the additions, deletions and unions is probably of small use.

Instead of tackling all these issues, the most simple solution has been chosen: Only classes can be selected, independent of the tree, package or cluster they appear in. Clicking on a package or cluster selects all its leaves.

To keep the expressive power of the selections, the user must be able to add arbitrary classes to the selection as well as remove classes from the selection. In addition, there should be some way of exclusively selecting a particular node, clearing the current selection and only selecting a chosen node. This mode would be used to navigate around quickly in the trees and collect information, in a similar way as a user may click through the files in a file browser to view their details.

So in addition to clicking on nodes, the user must also somehow specify what to do with the selected node – add, remove, or exclusively select. This could be implemented by providing three modes that can be toggled by keystrokes or buttons in a toolbar. This is already known from other applications and can be learned easily. However, the user needs to remember which mode they are in, and switching the modes is cumbersome if it needs to be done repeatedly.

Another option are modifier keys that have to be pressed to alter the behavior of a click. Popular desktop applications, including file browsers and vector graphic editors, use the control key to add items to the selection or remove them, if they are already selected. While this solution is not self-explaining (the user needs to be aware of the fact that pressing the control key changes behavior), it allows to work efficiently with the selection. No modes have to be remembered, no toolbar buttons pressed.

The control-key solution has been selected for this visualization. There is one special case: control-clicking on inner nodes that are partially selected, i.e. some of its leaves are selected and some are not. One valid behavior would be to invert the selection on a control-click. However, then there would be no way of selecting or deselecting partially selected nodes completely. Therefore, control-clicking a partially selected node first selects all the classes therein, making it a fully selected node which, on the second control-click, would be deselected completely.

In addition to the normal selection by clicking, nodes can be hovered with the mouse. This is a secondary selection that co-exists with normal selections. Its main purpose is assisting the selection process by allowing a quick brushing-and-linking functionality while preserving the main selection.

To represent these selection modes, four colors have to be chosen: unselected, selected, hovered but unselected as well as hovered and selected. In unselected state, classes are beige – a light color without qualitative associations that can still be distinguished from the grayscale of inner nodes. For the main selection, a blue color is used to

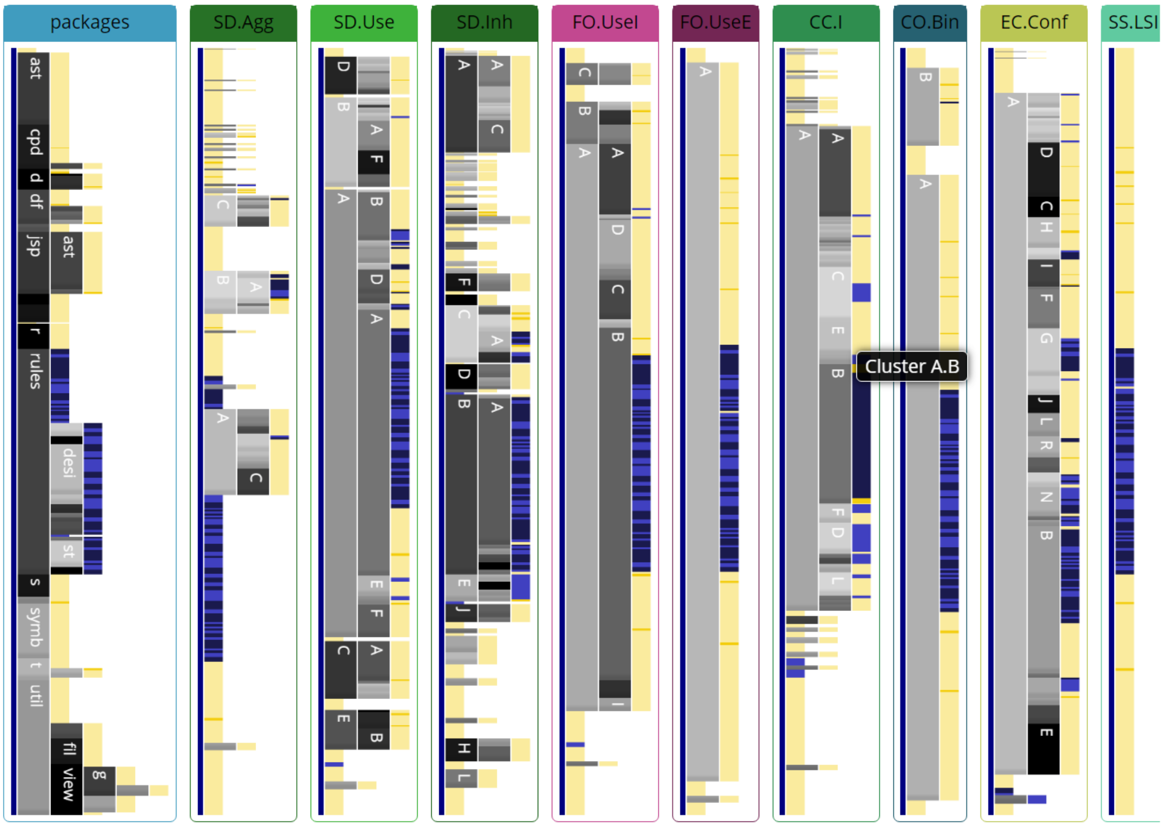


Figure 3.4: Brushing and Linking: The *rules* package has been selected by clicking, thus all classes of that package are highlighted in light or dark blue. Cluster *A.B* is hovered, so classes in this cluster are colored darker.

comply with most existing applications. The hover selection just darkens the node – if it is selected, the color is dark blue, if not, a darkened beige (see Figure 3.4).

The selection process can take a while. Clicking through the clusters and looking for classes that would fit in a concept involves several set operations. Options have to be considered, evaluated and then accepted or dismissed. To quickly get back to an earlier selection, a selection history has been implemented. Each operation – exclusive selection, addition or removal – pushes the selection to a stack; undo and redo functionality is provided to navigate through this stack. This feature is known and expected from most advanced applications.

In addition to the short-time undo history, it may also be desired to save selections for future reference or to share them with co-workers. This would be possible by capturing screenshots without a special functionality of the application. Then, however, the selection could not be changed afterwards and all interaction functionality like brushing and linking through hover or opening source code would no longer be possible. Therefore, a sharing and bookmarking feature is included in the application. As the reference implementation is a web application, this is implemented with URLs, see Section 4.3.1.

3.7 Node Ordering

The nodes of the package tree are ordered alphabetically to conform with the package views of IDEs. If a class name is known, it can be quickly found by scanning through the nodes (when the class names are not shown in the diagram itself due to lack of space, they are displayed as tooltip).

The clustering trees, however, have no inherent order. The classes in a cluster could be sorted alphabetically, but since the user does not even know in which cluster a class is, this is not really helpful. To find a class in a clustering tree, it should be looked up in the packages tree; then the location in other trees can be found using the brushing-and-linking technique described in Section 3.6.

Therefore, the ordering of the secondary hierarchies can be adjusted to improve the overall visualization by minimizing the distances between equivalent nodes. In this context, this means that each class should ideally be at the same vertical position across all trees. Of course, this can only be approximated.

The hierarchy should stay the same; the order of children of each leaf can be changed. In the end, each leaf node should be as close as possible to the corresponding leaf in the primary hierarchy (which is fixed). Ordering of leaf nodes is thus straightforward: they should occur in the same order as the ones in the primary hierarchy. For inner nodes, an algorithm has to determine the optimal ordering so that most leaf nodes can be positioned as well as possible.

The algorithm is based on the work by Sugiyama et al. for reducing edge crossing in graphs [STT81]. Holten et al. [HVW08] have applied the concepts to a hierarchy ordering. We use a similar approach, but can use a simpler algorithm because only one of the hierarchies needs to be sorted, the other one (the package structure) is fixed.

The idea is to position inner nodes at the *barycenter* of their leaf node's preferred position – or at least as close as possible to it. The barycenter can be calculated by Equation 3.2, where $p(l)$ stands for the preferred position of leaf node l.

$$bary(v) = \frac{\sum_{l \in \mathrm{leaves}(v)} p(l)}{|\mathrm{leaves}(v)|} \tag{3.2}$$

The sort algorithm traverses the tree recursively. It starts at the root node and sorts its direct children by their $bary$ value. Then it calls itself recursively on each child node.

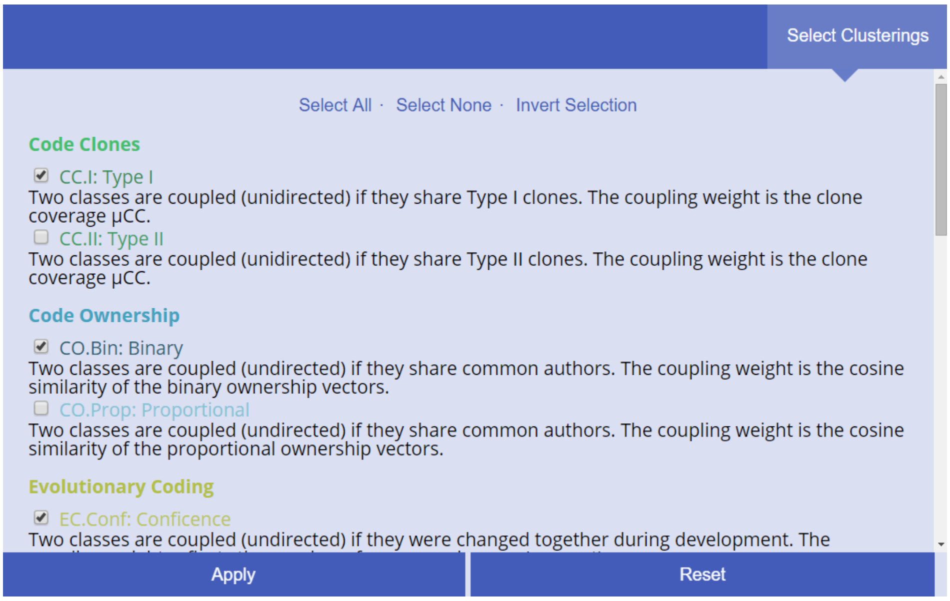
3.8 Clustering Selection

In our data set, there are 17 different clusterings for each project. Even more could be generated by adjusting parameters of the clustering algorithm. Comparing them all at once to the primary hierarchy would not only decrease the application's performance but also simply overwhelm the user with information. Therefore, the user can choose which clusterings should be shown as icicle plots. The button *Select Clusterings* toggles a list of all available clusterings that will be shown in the sidebar (see Figure 3.5). That way, the context of the currently shown clusterings is not lost, and the user can, e.g., remove a cluster from the content pane.

The clusterings are grouped by the categories introduced in Section 1.1. For each clustering, a short description is given explaining which classes are considered to be coupled. The keys that are shown in the icicle plot's headings are also displayed and have the same color so that users can more easily relate the items in the selection list to the icicle plots in the content pane.

The selection changes do not take effect immediately, but only after the user has pressed the *Apply* button. The main reason for this is performance: It takes a moment to calculate the similarity values of the new nodes and recalculate the values of the primary hierarchy; doing this after each checkbox tick would slow down the user. Additionally, if a mistake is made and then quickly corrected, the order of the icicle plots is not changed unintendedly.

In the selection pane, the clusterings have a fixed ordering so that, after having worked with the application for a while, the user gets accustomed to their location and can find and toggle them quickly even in new projects. In the content pane, however, the icicle plots can be rearranged. This allows users to compare two arbitrary hierarchies



Code Clones

☑ CC.I: Type I
Two classes are coupled (unidirected) if they share Type I clones. The coupling weight is the clone coverage µCC.

☐ CC.II: Type II
Two classes are coupled (unidirected) if they share Type II clones. The coupling weight is the clone coverage µCC.

Code Ownership

☑ CO.Bin: Binary
Two classes are coupled (undirected) if they share common authors. The coupling weight is the cosine similarity of the binary ownership vectors.

☐ CO.Prop: Proportional
Two classes are coupled (undirected) if they share common authors. The coupling weight is the cosine similarity of the proportional ownership vectors.

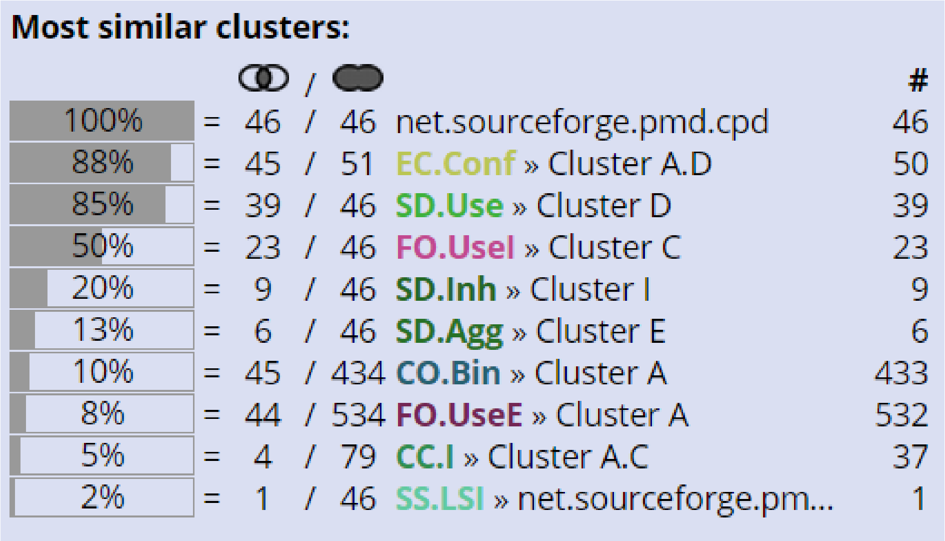
Evolutionary Coding

☑ EC.Conf: Conficence
Two classes are coupled (undirected) if they were changed together during development. The

Figure 3.5: Clustering Selection: Available clusterings are grouped by category and can be checked individually.

by putting them next to each other. It can also be seen as a secondary selection: If the most important clusterings are put next to the primary hierarchy, these can be inspected in detail while there are still some supporting icicle plots further away.

As mentioned in Section 3.6, the current selection can be bookmarked and shared. It would be desirable to include clustering selection and order would be included in this functionality so that the exact representation can be shared. However, for the undo feature, there are advantages as well as disadvantages. On the one hand, it would be nice and consistent to cover the whole selection data in the history, so that pressing the undo button returns the user to the previous view. On the other hand, tracking the selection history independently from the clustering enables another use case: reviewing previous selections with a different set of clusterings. For example, after having explored a few sets of classes, the user might decide to see if there would be more similar clusters if another clustering has been added. This is easier if the clustering selection is not included in the undo history as the selection can simply be changed, and then the undo history traversed to reload the past selections.



Most similar clusters:

			/			#
100%	=	46	/	46	net.sourceforge.pmd.cpd	46
88%	=	45	/	51	EC.Conf » Cluster A.D	50
85%	=	39	/	46	SD.Use » Cluster D	39
50%	=	23	/	46	FO.UseI » Cluster C	23
20%	=	9	/	46	SD.Inh » Cluster I	9
13%	=	6	/	46	SD.Agg » Cluster E	6
10%	=	45	/	434	CO.Bin » Cluster A	433
8%	=	44	/	534	FO.UseE » Cluster A	532
5%	=	4	/	79	CC.I » Cluster A.C	37
2%	=	1	/	46	SS.LSI » net.sourceforge.pm...	1

Figure 3.6: Similar Cluster List for a package of the *PMD* project

3.9 Similar Cluster List

Similarity information is encoded in node colors. This works well for an overview and as basis for discovery, but lacks in details. Brushing and linking allows users to identify classes across all hierarchies. However, it still does not provide a full explanation of a node's color. A node of a secondary hierarchy is dark if there is a package that contains mostly the same classes. In this case, it is easy enough to identify the corresponding package because most of its children are selected. The lower the similarity is, however, the harder it is to find the most relevant package. In the worst case, the user would have to manually count the selected classes. For the primary hierarchy, the problem is even more prominent: Selecting a package highlights various classes in all the secondary hierarchies. To find out which one is the most similar one, the user would have to count classes across the whole visualization. Again, for the extreme cases of strong similarity, this is possible, but at some point, an additional guide is required.

The list of similar clusters solves this problem. It is displayed in the sidebar to the right of the content pane and always shown when there is a selection. For each of the selected clusterings as well as for the primary hierarchy, it lists the cluster or package with the highest similarity to the selection set. If the user selected an inner node directly by clicking on it, that node is listed at the top because it has 100% similarity to the set of selected classes. For nodes of the primary hierarchy, the next item in the list is the clustering that determined the package's color coding.

Each entry in the list of similar clusters consists of five columns (see Figure 3.6). The first one is a bar chart visualizing the similarity value which is the Jaccard coefficient [AHS08] of the selection and the cluster. This is a value between 0% and 100%, so a bar chart that is filled accordingly provides a natural visual representation. Inside the bar chart, the percentage value is shown for detailed information.

Next to the bar chart, the absolute values of intersection and union are given. That is, the number of classes that occur in the selection and in the cluster, and the number of classes that are either in the selection or in the cluster. The quotient of these numbers is the definition of the Jaccard coefficient, so they provide more details and an explanation of the similarity value at the same time. In the list, these three columns are represented as mathematical equation: The similarity value is the intersection divided by the union.

The fourth column is simply the cluster name. For packages, this is the fully qualified package name; otherwise, the clustering name is given (colored the same way as icicle plot headings) followed by the qualified cluster identifier (e.g., *Cluster G.F.A*).

In the last column, the size of each cluster is given. This value could be derived, e.g., from the size of the union minus the number of selected classes. And for very similar clusters, the cluster size must be very similar to the selection size. Clusters that are further down in the list, however, can have one of three reasons for not being considered similar: either they contain too much classes (large union), too few classes (small intersection) or the wrong classes (small intersection and medium-sized union). The cluster size allows to differentiate between these as one value: a large number means too much classes, a low number too few classes and a value around the selection size identifies clusters with just wrong classes – each time assuming that the similarity value is low.

Hovering over an item highlights the icicle plot in the content pane by applying a drop shadow. Changing background color would be more prominent, but also interfere with other meanings of color in the visualization. This provides a further way of finding the icicle plot besides color coding of the clusterings. Additionally, hovering a list item triggers the hover selection: The classes in the hovered item will be painted slightly darker. As the main selection is kept, there are now four possible states for classes: Classes without selection (drawn in beige) are neither in the selection set, nor in the hovered cluster. Yellow items are not in the selection set either, but contained in the hovered cluster. Blue items are selected, but not in the hovered cluster. Dark blue items are in the intersection. That way, the intersection and union numbers can be explained.

3.10 Clustering Colors

The significance of unique colors for clusterings has been stated several times. They are required because clusterings occur in multiple places in the application – as headings of icicle plots, in the clustering selection and in the list of similar clusters. Colors provide a way to visually connect them. But this is not the only reason. There is actually information encoded in the colors: The color hue is determined by the clustering group. The groups are also used in the clustering selection (see Section 3.8) and used as prefix of clustering identifiers. Thus, the meanings of colors should become clear after having worked with the application for a while.

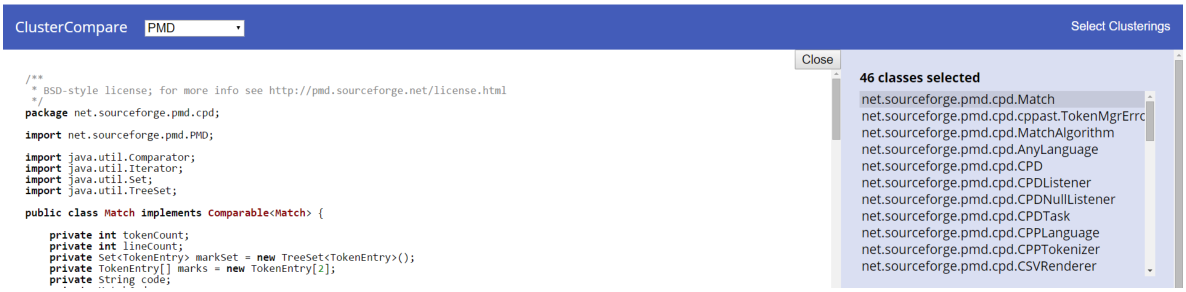
The mapping between clustering identifiers and colors should stay stable, even if clusterings are added in the future. This could be solved by a statically configured mapping file; however, that would not be flexible. Therefore, the color is derived from the clustering identifier using a hash function. Clustering group should be represented as hue, the individual clustering instance as brightness. The clustering identifiers always follow the syntax *group.instance*, so they are split at the dot. Then, both parts are hashed individually, the hash is converted to an integer and a modulus with the desired resolution is calculated (two bytes are more than sufficient for hue or brightness generation). Finally, the resulting value is mapped to the respective range – 0° to 360° for hue, and 25% to 75% for brightness.

The result is not ideal, because it is still possible that two different clustering groups are represented by the same or very similar hues, and the same can be said about clustering instances and their brightness values. For example, in the reference implementation, *CC* (code clones) and *SD* (structural dependencies) both use the green as base color, *CC*'s one being a little more blueish. However, the other four groups can be distinguished easily. In any case, hashing is a simple way to put semantic meaning into colors.

3.11 Source Code View

Although the general approach of the suggested visualization technique could be applied to hierarchy comparison in general, the purpose of this thesis is to put the visualization into the context of software modularization comparison. This is already done by using the terms *class* and *package*, labeling nodes with fully-qualified names and obviously by using sample data from software modularizations.

However, to integrate seamlessly into the process of software development and maintenance, more specifically into the refactoring phase, the user should be supported



```
/**
 * BSD-style license; for more info see http://pmd.sourceforge.net/license.html
 */
package net.sourceforge.pmd.cpd;

import net.sourceforge.pmd.PMD;

import java.util.Comparator;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class Match implements Comparable<Match> {

    private int tokenCount;
    private int lineCount;
    private Set<TokenEntry> markSet = new TreeSet<TokenEntry>();
    private TokenEntry[] marks = new TokenEntry[2];
    private String code;
```

46 classes selected

net.sourceforge.pmd.cpd.Match
net.sourceforge.pmd.cpd.cppast.TokenMgrErro
net.sourceforge.pmd.cpd.MatchAlgorithm
net.sourceforge.pmd.cpd.AnyLanguage
net.sourceforge.pmd.cpd.CPD
net.sourceforge.pmd.cpd.CPDListener
net.sourceforge.pmd.cpd.CPDNullListener
net.sourceforge.pmd.cpd.CPDTask
net.sourceforge.pmd.cpd.CPPLanguage
net.sourceforge.pmd.cpd.CPPTokenizer
net.sourceforge.pmd.cpd.CSVRenderer

Figure 3.7: The source code view as shown after clicking a class name

with additional information specific to the source code nature of these hierarchies. Full integration into the Integrated Developer Environment (IDE) would probably be the best approach. But it would also be the hardest to master completely: The user would expect the visualization to reflect source code changes and refactorings done in the IDE. Concluding, the whole clustering process has to be implemented in an IDE add-on. While this approach may be of interest to further studies in this area, the reference implementation in this thesis leaves out the IDE integration. See Section 4.1 for details.

To nevertheless provide a basic way of interacting with the presented software artifacts, a source code view exists. Clicking on a class in the selection pane shows its source code. It temporarily replaces the content pane with the icicle plots, so that the selection pane is still visible and the user can switch between classes. The source code view also supports syntax highlighting.

4 Implementation

This chapter presents the tool from a technical point of view. It shows how the visualization has been implemented, which technical design decisions have been made and what technical difficulties have arisen and how they have been treated.

4.1 Choice of Technology

For the choice of programming language and framework, there had been a few requirements:

- The framework should offer a solid graphics foundation. The software consists mainly of one visualization and it would be a waste of time to develop it on low-level graphic functions.
- The framework needs to be flexible. While the visualizations are based on existing work, they need to be customized to fit into the whole system. Full control over the appearance is required to experiment with new ideas.
- The language should allow fast prototyping. In the beginning of the project, only the most important aspects of the software had been specified. Many decisions had been made during the implementation phase. A language that allows to get started fast and to implement changes quickly is an important prerequisite for such an approach.
- Performance matters, because the user has to interact with the visualization. The application should feel responsive.

For software development related tools, the Eclipse platform is often used. It offers a cross-platform framework for creating user interfaces, accessing files and much more. By releasing a tool as a plug-in, it can be integrated into an existing IDE. This allows for a seamless workflow that involves interacting with the tool while navigating or editing the source code.

However, these advantages would be of little to no use for the tool developed for this thesis. The source code cannot be processed directly into the visualizations – a whole fleet of tools is needed to extract all the information necessary for the different coupling concepts, ranging from static analysis to extracting information from the source control. It would have been out of scope of this thesis to integrate all these tools into Eclipse. Instead, the data has been produced in advance and thus can be loaded in any environment.

While there are libraries for SWT (the UI foundation of Eclipse) that support drawing various charts (Prefuse, SWTChart, XYChart), they do not offer the flexibility needed. So either the foundation would have been missing, or the ability to change every detail of the presentation. In addition, Java tends to be rather slow to develop and minor adjustments can make up a substantial piece of code.

The alternative had been web technologies. A relatively new framework from 2011, called D3 (Data Driven Documents)[1] offers a foundation to develop all kinds of visualizations. It is built upon SVG (Scalable Vector Graphics) and thus natively supports basic shapes, bezier curves, gradients, shadows, animations and more. The visual elements can be bound to data and easily updated when the data changes.

Many examples demonstrate the capabilities of D3 and show how to implement certain visualizations. There is also one for Icicle Plots[2] implemented in less than 50 lines. This is possible because D3 already provides utility functions like color mapping, different scale types and for the actual drawing functions. The most important feature of D3 for this use case is the partition layout. It accepts a tree structure and calculates the coordinates of all nodes to form an Icicle Plot.

The logic is coded in JavaScript. It is a language in motion, at the time writing several useful features are in different stages of design and implementation. To make use of these features, the Babel transpiler (JavaScript to JavaScript compiler) is used that translates the new language constructs into equivalent pieces of JavaScript that is understood by current modern browsers.

Grunt is used as build tool. It watches the files and recompiles the application when a script or style file has been changed. It then updates the browser and reloads the page if necessary. This enables a fast change-and-review cycle which greatly eases the agile development process.

[1] http://d3js.org/

[2] http://bl.ocks.org/mbostock/1005873

This tool stack offers a graphics framework, is flexible and allows for fast prototyping, so it fulfills three of the four initial requirements. However, the performance aspect has to be evaluated. Browsers are known to be slow when there are lots of elements in one page, and the icicle implementation in D3 creates at least one element per class. Therefore, there will be a soft limit in the number of clusterings that can be shown simultaneously while still being responsive.

4.2 Data Format

To reduce the amount of work for the visualization tool, the data format has been chosen to be easily readable by JavaScript. For each project and within that project for each clustering plus for the original package structure, there is a JSON file accessible by a `GET` request. The contents of these JSON files is described as an example in listing 4.1.

While this format can be processed directly with JavaScript, it is still close enough to the original data generated by the clustering tool that this conversion process is straightforward and the main logic resides in the visualization tool and not in the converter. For this reason, there had not been need to change the converter's source code except for few bug fixes.

4.3 Architecture

The application roughly follows the Model-View-Controller principle, although not in the classic manner.

4.3.1 Model and ViewModel

There is only one entry point for all model data, the `ViewModel` class, which in turn contains an instance of the `Model` class. `Model` provides access to the persistent data, such as the list of projects, information about clusterings and the trees. Apart from this `Model` instance, `ViewModel` describes the application state, i.e. what is required to reconstruct what is currently visible on screen, given the persistent model data. The application state is serialized into the URL so that individual views can be bookmarked and shared as well as navigated through using the browser's back and forward buttons.

Listing 4.1 Tree Data Format

```
{
  project: "Name of the software project the classes are taken from",
  couplingConcept: "identifier of the coupling concpet, e.g. SD.Inh, or
       packages for original structure",
  clustering: {
    tool: "the name of the clustering tool used",
    summary: "short, human-readable information about the clustering
       tool/algorithm used"
  },
  root: {
    name: "display name for the node",
    qualifiedName: "complete class name, for leaves",
    children: [
      {
        name: "a child node",
        qualifiedName: "complete class name, for leaves",
        children: [
          { (node) }
        ]
      }
    ]
  }
}
```

The trees are not exposed exactly like they are represented in the Tree Data Structure. `NodeFactory` inspects the nodes and determines a class for each node. There are separate classes for `Class`, `Package`, `Cluster` and the `RootNode`. They are all subclasses of `Node` and override some of its methods and properties. That way, the presentation of nodes can be adjusted for each node type individually. For example, the label for package nodes is the package name, whereas the label for clusterings is a generated letter sequence denoting its index within the parent.

As previously stated, the `ViewModel` serializes its state into the URL. The selection state accounts for the largest amount of data because it takes one bit per class. Directly encoding this into the URL would not be practical. For example, the PMD data set contains more than 550 classes - that would be about 70 bytes, or 105 characters in base64 encoding. These urls would be rather inconvenient to share. However, the selections are normally not totally random, but tend to be in line with the package structure to some extend. This is obviously true for selecting packages but applies also to the most interesting clusters, those who have some resemblance to packages.

This means that when sorting the classes by their qualified names, the selections tend to roughly consist of one or multiple ranges. This is exactly what general purpose compressing algorithms are made for. Therefore, the deflate algorithm [Deu96] is used to compress the bit sequence of selected classes. Common selections can be stored in 50 to 60 characters (base64-encoded).

4.3.2 Visualization

To follow best practices in terms of low coupling and maintainability, the view module (for the visualization) has mostly been decoupled from the model module. While the view module obviously has to access the model for the data, it does not have to know how the data is received, organized and how, e.g., the similarity values are calculated. The model module does not even have to know the view at all.

The `Icicle` class, which is responsible for rendering an icicle plot, receives the tree to draw, the place where to put it, as well as a function mapping a node to its color value as parameters. The technology used for rendering the node is arbitrary and can be changed any time. This has been proved useful as will be seen in Section 4.4.

These icicles are wrapped in boxes called `VizItems` which decorate them with a title and border and allow the user to drag them. The items are grouped in one `VizContainer` which determines which trees to show from the view model and also synchronizes the ordering of the items with the view model.

4.3.3 Controllers

The modules introduced in the visualization section are a mixture of controller and view because they define the presentation and are also responsible for data binding and event handling. However, they are still decoupled in that they take the data providers as arguments. The `main` module acts as dependency injector: it instantiates the model, initializes the controllers, and connects both.

4.4 Rendering Performance

The sample data set contains projects from 99 classes (Cobertura) to 656 classes (JHotDraw) and 17 clusterings, thus up to 18 trees including the primary hierarchy. If all clusterings are selected, more than 5200 leaf nodes are to be drawn, not counting the inner nodes. The application should be responsive even when all clusterings are selected, so it has to cope with several thousands of nodes that are visible simultaneously. The interaction features described in Section 3.6 *Brushing and Linking* require the node background of any number of nodes to be changed when the user moves the mouse over the trees.

4.4.1 SVG

The traditional way of creating visualizations with D3 is by breaking it down into basic graphic elements like lines, rectangles or text labels. For each of these element, an SVG object is created, assigned with its properties and bound to data. An icicle plot, for example, consists in its most basic form of one rectangle per node plus a text element for each node that should be labeled.

Using SVG for icicle plot visualizations means that for each node in each tree, at least one SVG element is required. All these elements are managed by the browser which provides a set of features like event handling, styling with Cascading Style Sheets or inspection in the developer console. This overhead is not neglectable and can result in a significant performance loss if several thousands of elements are loaded in the document.

In the first iteration, this SVG approach has been used. If only two or three clusterings are selected, the application still feels responsive, although the frame rate regularly drops below 30 frames per seconds in Chrome on a modern desktop computer. When ten clusterings are displayed at once, every action including node hovering, clustering reorder or horizontal scrolling in the content pane feels sluggish with a constant frame rate between five and ten frames per second.

4.4.2 Throttle

This is especially a problem for the brushing and linking functionality: Moving the mouse vertically down on an icicle plot triggers a mouse enter and leave event for each node in that layer. These events are queued by the browser and handled one after the other. Each time, the nodes to be highlighted, i.e., the classes those in the hovered cluster in all trees, have to be computed and their background color changed. When this process takes longer than the user to move over the nodes, the highlighting lags behind the cursor movement and the application feels very unresponsive.

To reduce this problem, the hover events are *throttled*. This means that not every hover event is processed directly, but only the last one of one continuous cursor movement. If the user moves the mouse quickly over the chart, nothing happens. Just when they take a short break over one node, this node is considered hovered and the highlighting adjusted. There are no events queued so the reaction is as quick as possible. This is implemented by deferring the event handler for a set amount of time, for example, 100 milliseconds. Each time a new event occurs, the last deferred event handler is canceled and the new one scheduled for 100 milliseconds from the time of its occurrence. That way, always the last event is scheduled, and it only is handled when there is no event for 100 milliseconds.

4.4.3 Canvas Rendering

However, the problem of overall low performance and slow reaction times is not solved by this trick. Therefore, an alternative implementation for icicle plot rendering has been tried.

Apart from SVG documents, the Web provides another technology for drawing with basic shapes: The 2D canvas [CWG+14]. The features are comparable as it also supports drawing rectangles, paths and text. The concept is different though: In SVG, graphic elements are created as objects and then kept so that they can be manipulated afterwards and scaled as required. Using canvas, graphics are drawn imperatively directly on the drawing area and apart from the pixel colors, no information is persisted. When data changes or the window has been resized, the graph or parts of it have to be redrawn manually.

On the one hand, this shifts the responsibility of redrawing existing but changed structures efficiently to the application code, so optimizations on the browser vendor's side are more limited in their nature. For example, in SVG documents, the browser could decide to not draw elements that are completely covered by a different element; this logic would have to be implemented manually in the application code using canvas. On the other hand, SVG documents can get very complex, so when optimizing a SVG renderer, many corner cases have to be taken into account. Especially for simple visualizations, manual optimization are easier to implement.

In our case, the large number of elements is a problem with the SVG approach, therefore the performance is low even if the icicles do not change at all, e.g., when scrolling the contents pane or reordering clusterings. Using the canvas approach, there are no persistent objects so these interactions are not impacted by the number of classes.

Interaction via brushing and linking requires parts of the icicle plot being redrawn on cursor movement. This whole task can be greatly optimized with the canvas approach. Instead of registering thousands of event listeners, each tree only requires one *mousemove* and one *mouseleave* listener. In this listener, the node under the cursor is calculated by applying the scale functions used for drawing the rectangles *in reverse* and then traversing the tree from root to the hovered layer. For each level, the child node under the cursor is determined by comparing the vertical coordinate to the known vertical position of each child node.

Updating the colors of an icicle plot when the main or hover selection changes is faster, too: No expensive DOM manipulation is required, only the rectangles of nodes whose selection status changed have to be repainted.

Both alternatives – the SVG as well as the canvas approach – have been implemented and thus can be compared directly. That way, the superiority of the canvas method is evident: Even with all clusterings selected, on a modern desktop computer interactions such as node reordering or brushing and linking respond without noticeable delay. Therefore, the canvas approach has been chosen in the final implementation.

4.5 Data Gathering

The visualization is mostly agnostic to the data it represents. Basically, a set of classes is required that is organized in some kind of package or namespace structure. For each secondary hierarchy, a coupling concept has to be defined – a matrix indicating which classes are related. This matrix is used as input of a clustering algorithm which produces a tree of clusters.

The data used for the illustrations in this thesis have been gathered by Beck and Diehl [BD11]. They provided a Java library for reading the coupling data as graphs. This library has been used and extended to an automatic tool, called *cluster converter*, that reads the graphs, calls an external clustering program and finally saves the resulting trees in individual files.

Rosvall's and Bergstrom's tool *infohiermap* [RB11] has been used to cluster the graphs. It is written in C++ and can be compiled on linux via a simple *make* command. The resulting binary can be executed with arguments to specify source and target files as well as options. It generates a clustered tree in a text file format with the *.tree* extension. The *.tree* files list each leaf in one line with three columns: the hierarchy path (a colon-separated list of intermediate nodes from the root to the leaf node), a weight value and the node identifier. Inner nodes are not represented by text lines but can only be derived from the hierarchy path of the leaf nodes they contain. This format is not ideal for processing in a JavaScript application, so the cluster converter tool also converts the *.tree* files to the JSON format described in Section 4.2.

The tool only has be be run once per data, then all information are available as static files. Therefore, no back-end is required for the visualization application; it can be served from a standard web server like Apache[3].

The source code view feature of Section 3.11 downloads source code on demand and expects the files to be in a subdirectory of the project directory. They have to be copied to this location in order to complete a set of data that can be processed by the visualization application.

4.5.1 Data Preprocessing

Apart from format conversions, the data generated by the clustering program can mostly be used as-is, with one notable exception. The clustering algorithm never generates clusters that contain both subclusters and leaf nodes. Leaf nodes that would occur besides inner nodes are therefore wrapped in their own subcluster with just the one child node. In the visualization, those leaf nodes would be displayed one level deeper than they really are. There is also no semantic significance of this behavior.

Thus, nodes containing only one child are unwrapped before being processed. To keep the cluster converter tool as simple as possible, this logic has been implemented in the JavaScript part. The `NodeFactory` class which converts the JSON input into a tree structure with the correct classes tackles this issue. Before even creating the node instances, it iterates over each node and replaces nodes that have only one child with exactly this child.

[3] https://httpd.apache.org

5 Evaluation

In this final chapter, the visualization will be tested in a small case study using two Java projects. The study has been performed by the author of this thesis and thus might be biased towards people who already know the visualization well. The purpose of the study is not to find out how fast users can learn to work with the visualization, but rather to determine whether the clustering data can effectively be analyzed by a user already familiar with the visualization.

To maximize possible findings, all clusterings have initially been selected for both projects.

5.1 Example: PMD

PMD[1] is a source code analyzer for several languages including Java, JavaScript and XML. It uses a set of rules to find common programming mistakes like unused variables or dead code.

The PMD package structure is relatively flat; most packages are located directly under the `net.sourceforge.pmd` package and the maximum depth after this root package is four for `net.sourceforge.pmd.util.viewer.gui.menu`. As seen in Figure 5.1, the top two thirds of the first level packages have dark colors; clicking on the lighter ones of these reveals that with one exception they all have a corresponding cluster of at least 70% similarity. The most similar clusters for the lower third has between 25% and 37% similarity.

To get an overview of the secondary hierarchies, they have been selected all at first. This shows many points of interest. Especially *SD.Use*, *SD.Inh* as well as the two *EC* clusterings have several clusters with high similarity values. However, there are also useless clusterings like *FO.UseE* or *SS.Tfidf* which mainly consist of one big cluster. They have been removed in the following analysis to focus on the most promising

[1] https://pmd.github.io/

clusterings. The new ordering is *SD.Use, SD.Agg, SD.Inh, EC.Conf, EC.Sup, CC:I, CC.II, FO.UseI, FO.InhI, FO.AggE, FO.AggI*.

5.1.1 Inspecting one Package

Interestingly, while the lower third of the first level has overall low similarity values, at the second level, there packages with better matches, `util.viewer` being the best among them. It has two perfect matches, in *EC.Sup* and *EC.Conf* each (see Figure 5.2). This suggest the classes of this package have often been modified together.

Next up are two clusterings of the *Structural Dependencies* group at 60% and 53%. The percentages are rather low, but looking at the other columns shows that they are approximate subsets of the `util.viewer` package: *SD.Use* shares 12 classes with the package and is 13 classes in size, so it contains one additional class. *SD.Agg* even is a perfect subset with 10 classes. By clicking on the *SD.Use* row, it can be confirmed that *SD.Agg* is in turn a subset of that clustering. So these 10 classes have really strong evidence for being in one package. Using the back button of the browser, the previous selection is restored.

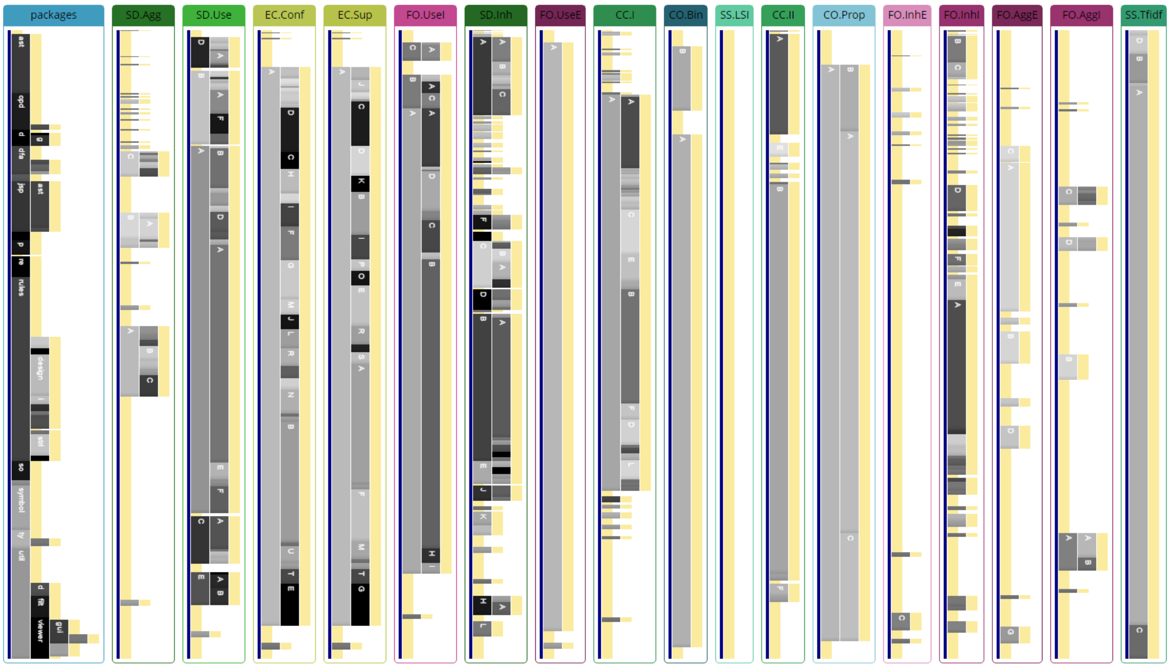
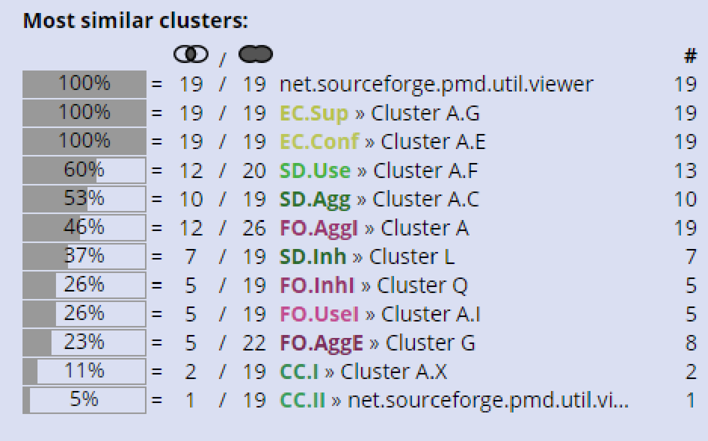


Figure 5.1: Evaluation: PMD Project



Most similar clusters:

			/			#
100%	=	19	/	19	net.sourceforge.pmd.util.viewer	19
100%	=	19	/	19	**EC.Sup** » Cluster A.G	19
100%	=	19	/	19	**EC.Conf** » Cluster A.E	19
60%	=	12	/	20	**SD.Use** » Cluster A.F	13
53%	=	10	/	19	**SD.Agg** » Cluster A.C	10
46%	=	12	/	26	**FO.AggI** » Cluster A	19
37%	=	7	/	19	**SD.Inh** » Cluster L	7
26%	=	5	/	19	**FO.InhI** » Cluster Q	5
26%	=	5	/	19	**FO.UseI** » Cluster A.I	5
23%	=	5	/	22	**FO.AggE** » Cluster G	8
11%	=	2	/	19	**CC.I** » Cluster A.X	2
5%	=	1	/	19	**CC.II** » net.sourceforge.pmd.util.vi...	1

Figure 5.2: List of most similar clusters of package `util.viewer`

The next question is which classes are *not* included in the *SD* clusterings. It can be answered by clicking on the cluster *A.F* in the visualization two times with the Control key pressed down. Most of the classes are unclustered, and the remaining two occur in a subcluster of *A* and thus in the same top-level cluster. This cluster, however, is only light gray and with the hover action it quickly becomes clear that it has no support in the primary nor in any of the secondary hierarchies.

5.1.2 Combining two Packages

Cluster *A.A* is the darkest of *CC.I*, and selecting it shows that it contains a majority of the classes of `ast` and `jsp.ast`. The list of most similar clusters states 68% as the similarity, but one expects this value to be higher if both AST-related packages would be considered together. This assumption is verified by clicking selecting both packages via Control-click: The new similarity is 77%.

5.2 Example: Wicket

Apache Wicket[2] is a component-based Java web framework. Again, clusterings with few clusters or extremely low similarity values have been excluded, and they are exactly the same like in the PMD data set.

[2] `https://wicket.apache.org/`

Wicket is structured in many small packages, the majority having fewer than ten classes. Throughout the whole hierarchy, most packages have good matches with similarity values above 80%. The values of the largest four high-level packages, covering 470 classes, have similarity values below 35%, though.

5.2.1 Converter Classes

Wicket contains a few classes for converting values like numbers and dates to strings and back. Those are organized in the package `util.convert`. The whole package is backed by the two Evolutionary Coding clusterings with just four classes missing. Two of the classes of `util.convert` that are not in these clusterings are indeed not directly related to converters: `ILocalizable` annotates a class that needs localization information, and `LocalizableAdapter` is a simple implementation of this interface (they are used by converters though).

Cluster F of *SD.Inh* almost exactly matches the `converters` subpackage which is obvious because all converters inherit from `AbstractConverter`. The cluster also contains the interface `IConverter`. Although it is not part of the class hierarchy, all converters implement this interface, so it is correct for it to occur in the cluster. The developers of

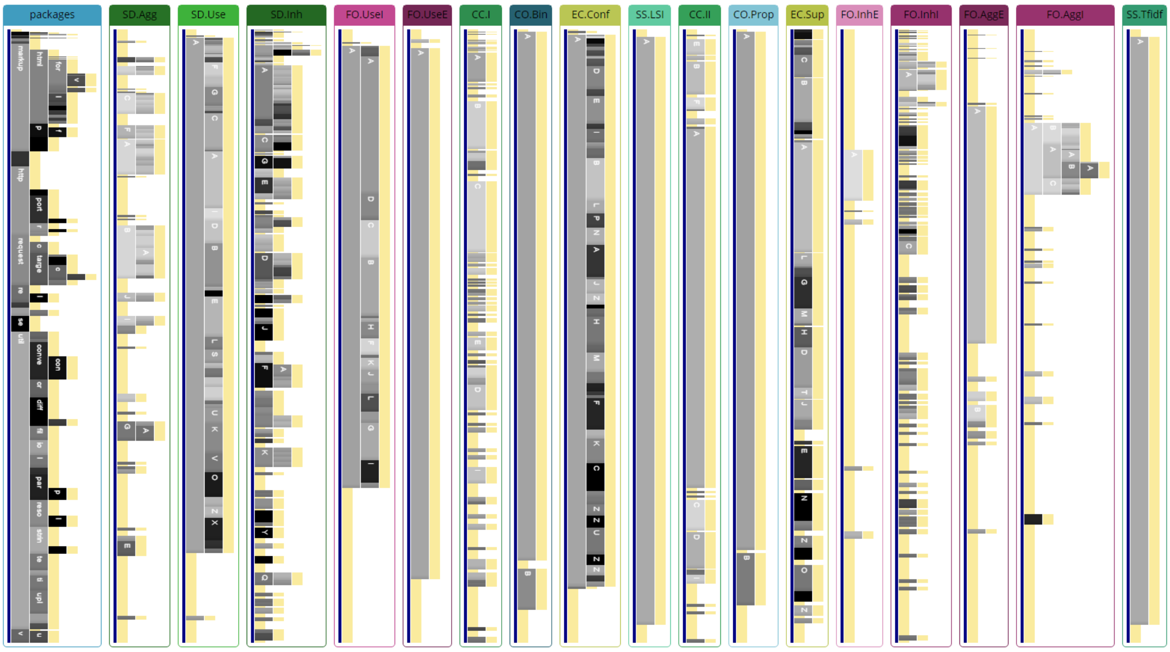


Figure 5.3: Evaluation: Apache Wicket

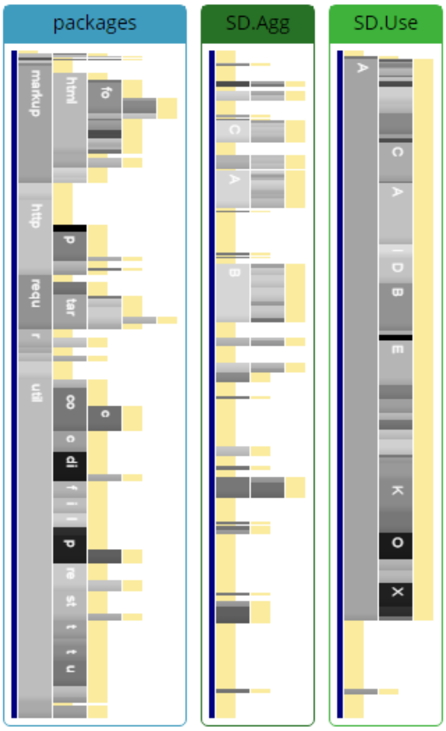


Figure 5.4: The two clusterings SD.Use and SD.Agg have relatively low similarity values.

Wicket have not included it in the `converters` package. There is nothing wrong with this decision, especially because of the package name, but it is noteworthy anyway.

5.2.2 Low Coupling, High Cohesion

The coupling concepts *SD.Use* and *SD.Agg* are most related to the principle of Low Coupling and High Cohesion. Thus, in a good package structure, there should be many matches between packages and clusters of those clusterings.

However, having only these two clusterings selected, the icicle plots are colored rather light (see Figure 5.4. Apart from two exceptions of 67% and 69%, which can be easily spotted, the clusters of *SD.Agg* have similarity values of 50% and below (see Figure 5.4.

SD.Use looks better: It contains four clusters of 80% similarity and higher. Two of them belong to the same package as the two top clusters of *SD.Agg* which can be both seen in the list of similar clusters and by the brushing and linking highlighting.

6 Conclusion

In this thesis, a visualization has been developed that compares the package structure of a software project to other hierarchies of the same classes, generated by clustering algorithms. Icicle plots are used for the individual hierarchies as a compact presentation. Clusterings can be selected and shown side by side to the main package structure in a small multiples diagram. Color coding conveys the information of node similarity between all the hierarchies. These general design decisions have been discussed and explained in Chapter 3 up to Section 3.4.

Afterwards, extensions were made that enhance the comparison process and allow to generate more insights. Brushing and Linking (Section 3.6) as the main means of interaction is coupled with a cluster list (Section 3.9). Secondary aspects of the visualization like node ordering or border visualization have been considered, too.

As documented in Chapter 4, the visualization is implemented as a web application. Modularization and clustering data of 17 open source projects have been converted to the tool's custom data format and are included in the application. In a small case study in Chapter 5, the visualization has been evaluated on two open source projects.

6.1 Evaluation Results

The icicle plots of individual hierarchies visualize the trees in a suitable manner: A whole software project can be displayed multiple times at once on a normal computer screen. Especially for developers that are not familiar with a project, having the complete package structure visible at once gives a good overview. Of course, class names are not visible at once, but they are not required for a first overview. Yet, the tooltip functionality that displays class or package names on demand is useful to get an idea of which classes are in a given package.

The node coloring reveals at a first glance how similar the chosen software project is to the clustering results. If there are many black or dark gray rectangles, the chances are high that useful matches are to be found. Uninteresting clusterings, for example, flat ones or trees with very low similarity values can be dismissed and removed quickly. Reordering the remaining hierarchies allows to put the focus on the most promising clusterings.

On a second glance, interesting nodes can be identified again by considering color. Dark nodes with high similarity obviously stand out, but it is also possible to find those with medium or low similarity values by keeping the specific color in mind and scanning the hierarchy vertically.

The node ordering algorithm works well: If multiple clusterings have clusters that are very similar to each other, they occur roughly on one horizontal line. This allows to quickly identify clusters that are backed by multiple clusterings. This initial assumption can be verified by hovering over one of the packages and to see if the other packages are highlighted as well.

The list of most similar clusters can be used to derive several insights. The percentage bars show the similarity distribution in form of a histogram. This supplements the node color with the information if there is only one or several nodes with that similarity value. The other numerical columns show whether the similarity value is decreased due to missing classes or additional classes in the clustering. It could however be considered to write the numbers of additional and missing classes in the table so that the user does not have to calculate those themselves. Overall, the application feels very responsive, most reactions to mouse movements and clicks are instant on a modern computer[1].

The visualization gives a good overview and entry point into a package structure. It not only presents the actual package structure but also explains it: Depending on which clustering has high similarity, it can be derived which of the principles introduced in Section 1.1 are followed in which packages. That way, developers new to a project are assisted in figuring out where to put new classes.

However, to actually suggest well-founded restructurings, the visualization alone does not suffice. It is still necessary to know the source code in order to fully understand the reasoning behind each package. But the visualization can support developers in the restructuring task by highlighting packages of interest and providing possible explanations for them.

[1] Clicking on a cluster containing several hundreds of classes involves a slight delay.

6.2 Future Work

During the development of this visualization tool, many promising ideas came up that had to be postponed to future versions due to of lack of time. This section glances briefly over the suggestions and explains why they would be useful.

6.2.1 Matrix View

While the icicle plots are a good visualization of clustering results, they do not convey the underlying data in its details. It may be desirable to determine why exactly a set of classes have been clustered. Therefore, the user may wish to see the raw coupling matrix. It is essentially a directed or undirected graph with all the classes and coupling values on the edges. A node-link diagram would not be suitable because of the high number of classes and especially the high connectivity. Instead, the matrix could be presented as a table with class names at the horizontal and vertical axis.

This matrix view should be accessible from the clusterings. It would also be possible to highlight clusters in the matrix view by changing the background color of all contained classes, so an entry from a cluster node would also be possible.

6.2.2 Comparing Selections

It is possible to select a set of arbitrary classes which then can be inspected in the list of similar clusters. This idea could be extended to a comparison between two selections: The user would save a selection to a temporary place, make a new one and press a button to compare those. Then, a Venn diagram [Ven80] would be shown visualizing how many and which classes are contained in both selections compared to the classes that occur just in one selection.

This feature would serve as a zoom and focus functionality. When one cluster (or several combined) is to be compared to an existing package, the complete icicle plots are not ideal because they contain many unrelated classes. The classes are still only few pixels high in the icicle plots, so it can sometimes be hard to see which class is selected. This feature would limit the visualization to the two selections of interest and thus provide better visibility.

6.2.3 Alternative Similarity Aggregations

The coloring of nodes in the secondary hierarchy is straightforward: It depicts the similarity of the most similar node in the primary hierarchy. The coloring of nodes in the primary hierarchy, however, has to be some kind of aggregation: There are multiple secondary hierarchies which all have one most similar cluster with its own similarity value. Currently, the maximum of those values has been chosen for the coloring. This makes sure no package with a good match is missed.

Sometimes, it would be good to know how many clusterings have similar clusters to a package. This question can roughly be answered by looking for clusters at the same vertical position, as has been illustrated in Section 6.1. Showing this information in the package nodes themselves would however be much more visible.

Therefore, a possible extension would be to let the user select an aggregation function. The default can still be the maximum. Average and minimum would be sensible only if very few clusterings have been selected and there is high correlation between all of the clusterings and the package structure. However, an aggregation like *second best* could still be applied to many clusterings and provide the useful information of which packages are backed with high similarity to more than one clustering.

6.2.4 Two-Hierarchy Comparison

In Section 2.3 of the Related Work chapter, Holten's and Van Wijk's two-tree comparison [HVW08] stand out as a especially suitable visualization for comparing two package hierarchies. It does not scale well to a multi-hierarchy comparison, but could serve as extension to the small multiple visualization. The core idea is to draw bundled edges between two icicle plots. One of them has to be mirrored so that they face each other. This feature could be seamlessly added to the visualization: When the user drags one icicle plot *onto* another one, they could be selected for dual comparison. Then, the right one would be mirrored and bundled edges drawn between them.

Bibliography

[AHS08] P. Achananuparp, X. Hu, X. Shen. The Evaluation of Sentence Similarity Measures. In *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '08, pp. 305–316. Springer-Verlag, Berlin, Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-85836-2_29`. (Cited on page 38)

[AK02] N. Amenta, J. Klingner. Case study: Visualizing sets of evolutionary trees. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pp. 71–74. IEEE, 2002. (Cited on page 15)

[BC87] R. A. Becker, W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987. URL `http://dx.doi.org/10.2307/1269768`. (Cited on page 31)

[BD11] F. Beck, S. Diehl. On the Congruence of Modularity and Code Coupling. In *SIGSOFT/FSE '11 and ESEC '11: Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*, pp. 354–364. ACM, 2011. URL `http://dx.doi.org/10.1145/2025113.2025162`. (Cited on pages 10, 11, 12 and 48)

[BRW10] M. Burch, M. Raschke, D. Weiskopf. Indented pixel tree plots. In *Advances in Visual Computing*, pp. 338–349. Springer, 2010. (Cited on page 23)

[BWB$^+$14] F. Beck, F.-J. Wiszniewsky, M. Burch, S. Diehl, D. Weiskopf. Asymmetric Visual Hierarchy Comparison with Nested Icicle Plots. In *Joint Proceedings of the Fourth International Workshop on Euler Diagrams and the First International Workshop on Graph Visualization*, pp. 53–62. 2014. (Cited on page 28)

[Cal84] T. C. Callaghan. Dimensional interaction of hue and brightness in preattentive field segregation. *Perception & Psychophysics*, 36(1):25–34, 1984. URL `http://dx.doi.org/10.3758/BF03206351`. (Cited on page 29)

[Cho10] G. Chowdhury. *Introduction to Modern Information Retrieval, Third Edition*. Facet Publishing, 3rd edition, 2010. (Cited on page 12)

[Con68] M. E. Conway. How do committees invent? *Datamation Journal*, 14(4):28–31, 1968. (Cited on page 12)

[CWG+14] R. Cabanier, T. Wiltzius, E. Graff, I. Hickson, J. Munro. HTML Canvas 2D Context. Last call WD, W3C, 2014. URL http://www.w3.org/TR/2014/WD-2dcontext-20140520/. (Cited on page 47)

[Deu96] L. P. Deutsch. DEFLATE compressed data format specification version 1.3. 1996. (Cited on page 45)

[GK05] M. Graham, J. Kennedy. Extending taxonomic visualisation to incorporate synonymy and structural markers. *Information Visualization*, 4(3):206–223, 2005. (Cited on page 18)

[GK10] M. Graham, J. Kennedy. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9(4):235–252, 2010. URL http://dx.doi.org/10.1057/ivs.2009.29. (Cited on pages 15, 22 and 25)

[HVW08] D. Holten, J. J. Van Wijk. Visual comparison of hierarchically organized data. *Computer Graphics Forum*, 27(3):759–766, 2008. (Cited on pages 17, 34 and 60)

[ID90] A. Inselberg, B. Dimsdale. Parallel Coordinates: A Tool for Visualizing Multi-dimensional Geometry. In *Proceedings of the 1st Conference on Visualization '90*, VIS '90, pp. 361–378. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. (Cited on page 26)

[JS91] B. Johnson, B. Shneiderman. Tree-Maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pp. 284–291. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991. (Cited on page 24)

[KL83] J. Kruskal, J. Landwehr. Icicle Plots: Better displays for hierarchical clustering. *The American Statistican*, 37(2), 1983. (Cited on page 24)

[Mac86] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Trans. Graph.*, 5(2):110–141, 1986. URL http://dx.doi.org/10.1145/22949.22950. (Cited on page 28)

[MGT+03] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, Y. Zhou. TreeJuxtaposer: Scalable Tree Comparison Using Focus+Context with Guaranteed Visibility. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pp. 453–462. ACM, New

York, NY, USA, 2003. URL http://dx.doi.org/10.1145/1201775.882291. (Cited on page 15)

[Par71] D. L. Parnas. Information distribution aspects of design methodology. *IFIP Congress*, 1971. (Cited on page 11)

[Par72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972. URL http://doi.acm.org/10.1145/361598.361623. (Cited on page 11)

[PLCB04] C. S. Parr, B. Lee, D. Campbell, B. B. Bederson. Visualizations for taxonomic and phylogenetic trees. *Bioinformatics*, 20(17):2997–3004, 2004. (Cited on page 16)

[RB11] M. Rosvall, C. T. Bergstrom. Multilevel Compression of Random Walks on Networks Reveals Hierarchical Organization in Large Integrated Systems. *PLoS ONE*, 6:18209, 2011. URL http://dx.doi.org/10.1371/journal.pone.0018209. (Cited on page 48)

[RC07] C. K. Roy, J. R. Cordy. A Survey on Software Clone Detection Research. *Technical Report, Queen's University*, 115, 2007. (Cited on page 12)

[Sch91] R. W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pp. 83–92. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991. (Cited on page 11)

[SMC74] W. P. Stevens, G. J. Myers, L. L. Constantine. Structured Design. *IBM Syst. J.*, 13(2):115–139, 1974. URL http://dx.doi.org/10.1147/sj.132.0115. (Cited on page 10)

[STT81] K. Sugiyama, S. Tagawa, M. Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981. (Cited on page 34)

[TA08] A. Telea, D. Auber. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, 2008. (Cited on page 18)

[Tuf83] E. R. Tufte. *The visual display of quantitative information*. Graphics Press, 1983. (Cited on page 26)

[Ven80] J. Venn. I. On the diagrammatic and mechanical representation of propositions and reasonings. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 10(59):1–18, 1880. (Cited on page 59)

All links were last followed on October 10, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature